reinterpret_cast 中的一种。dynamic_cast 支持运行时类型识别，我们将在 19.2 节（第 730 页）对其做更详细的介绍。*cast-name* 指定了执行的是哪种转换。

static_cast

任何具有明确定义的类型转换，只要不包含底层 const，都可以使用 static_cast。例如，通过将一个运算对象强制转换成 double 类型就能使表达式执行浮点数除法：

```
// 进行强制类型转换以便执行浮点数除法
double slope = static_cast<double>(j) / i;
```

当需要把一个较大的算术类型赋值给较小的类型时，static_cast 非常有用。此时，强制类型转换告诉程序的读者和编译器：我们知道并且不在乎潜在的精度损失。一般来说，如果编译器发现一个较大的算术类型试图赋值给较小的类型，就会给出警告信息；但是当我们执行了显式的类型转换后，警告信息就会被关闭了。

static_cast 对于编译器无法自动执行的类型转换也非常有用。例如，我们可以使用 static_cast 找回存在于 void* 指针（参见 2.3.2 节，第 50 页）中的值：

```
void* p = &d;    // 正确：任何非常量对象的地址都能存入 void*
//正确：将 void* 转换回初始的指针类型
double *dp = static_cast<double*>(p);
```

当我们把指针存放在 void* 中，并且使用 static_cast 将其强制转换回原来的类型时，应该确保指针的值保持不变。也就是说，强制转换的结果将与原始的地址值相等，因此我们必须确保转换后所得的类型就是指针所指的类型。类型一旦不符，将产生未定义的后果。

const_cast

const_cast 只能改变运算对象的底层 const（参见 2.4.3 节，第 57 页）：

```
const char *pc;
char *p = const_cast<char*>(pc); // 正确：但是通过 p 写值是未定义的行为
```

对于将常量对象转换成非常量对象的行为，我们一般称其为“去掉 const 性质（cast away the const）”。一旦我们去掉了某个对象的 const 性质，编译器就不再阻止我们对该对象进行写操作了。如果对象本身不是一个常量，使用强制类型转换获得写权限是合法的行为。然而如果对象是一个常量，再使用 const_cast 执行写操作就会产生未定义的后果。

只有 const_cast 能改变表达式的常量属性，使用其他形式的命名强制类型转换改变表达式的常量属性都将引发编译器错误。同样的，也不能用 const_cast 改变表达式的类型： 164

```
const char *cp;
// 错误：static_cast 不能转换掉 const 性质
char *q = static_cast<char*>(cp);
static_cast<string>(cp);      // 正确：字符串字面值转换成 string 类型
const_cast<string>(cp);       // 错误：const_cast 只改变常量属性
```

const_cast 常常用于有函数重载的上下文中，关于函数重载将在 6.4 节（第 208 页）进行详细介绍。

reinterpret_cast

reinterpret_cast 通常为运算对象的位模式提供较低层次上的重新解释。举个例

子，假设有如下的转换

```
int *ip;
char *pc = reinterpret_cast<char*>(ip);
```

我们必须牢记 pc 所指的真实对象是一个 int 而非字符，如果把 pc 当成普通的字符指针使用就可能在运行时发生错误。例如：

```
string str(pc);
```

可能导致异常的运行时行为。

使用 reinterpret_cast 是非常危险的，用 pc 初始化 str 的例子很好地证明了这一点。其中的关键问题是类型改变了，但编译器没有给出任何警告或者错误的提示信息。当我们用一个 int 的地址初始化 pc 时，由于显式地声称这种转换合法，所以编译器不会发出任何警告或错误信息。接下来再使用 pc 时就会认定它的值是 char*类型，编译器没法知道它实际存放的是指向 int 的指针。最终的结果就是，在上面的例子中虽然用 pc 初始化 str 没什么实际意义，甚至还可能引发更糟糕的后果，但仅从语法上而言这种操作无可指摘。查找这类问题的原因非常困难，如果将 ip 强制转换成 pc 的语句和用 pc 初始化 string 对象的语句分属不同文件就更是如此。

reinterpret_cast 本质上依赖于机器。要想安全地使用 reinterpret_cast 必须对涉及的类型和编译器实现转换的过程都非常了解。

165

建议：避免强制类型转换

强制类型转换干扰了正常的类型检查（参见 2.2.2 节，第 42 页），因此我们强烈建议程序员避免使用强制类型转换。这个建议对于 reinterpret_cast 尤其适用，因为此类类型转换总是充满了风险。在有重载函数的上下文中使用 const_cast 无可厚非，关于这一点将在 6.4 节（第 208 页）中详细介绍；但是在其他情况下使用 const_cast 也就意味着程序存在某种设计缺陷。其他强制类型转换，比如 static_cast 和 dynamic_cast，都不应该频繁使用。每次书写了一条强制类型转换语句，都应该反复斟酌能否以其他方式实现相同的目标。就算实在无法避免，也应该尽量限制类型转换值的作用域，并且记录对相关类型的所有假定，这样可以减少错误发生的机会。

旧式的强制类型转换

在早期版本的 C++语言中，显式地进行强制类型转换包含两种形式：

```
type (expr);          // 函数形式的强制类型转换
(type) expr;          // C 语言风格的强制类型转换
```

根据所涉及的类型不同，旧式的强制类型转换分别具有与 const_cast、static_cast 或 reinterpret_cast 相似的行为。当我们在某处执行旧式的强制类型转换时，如果换成 const_cast 和 static_cast 也合法，则其行为与对应的命名转换一致。如果替换后不合法，则旧式强制类型转换执行与 reinterpret_cast 类似的功能：

```
char *pc = (char*) ip; // ip 是指向整数的指针
```

的效果与使用 reinterpret_cast 一样。

与命名的强制类型转换相比，旧式的强制类型转换从表现形式上来说不那么清晰明了，容易被看漏，所以一旦转换过程出现问题，追踪起来也更加困难。

4.11.3 节练习

练习 4.36：假设 i 是 int 类型，d 是 double 类型，书写表达式 i*=d 使其执行整数类型的乘法而非浮点类型的乘法。

练习 4.37：用命名的强制类型转换改写下列旧式的转换语句。

```
int i; double d; const string *ps; char *pc; void *pv;
(a) pv = (void*)ps;    (b) i = int(*pc);
(c) pv = &d;           (d) pc = (char*) pv;
```

练习 4.38：说明下面这条表达式的含义。

```
double slope = static_cast<double>(j/i);
```

4.12 运算符优先级表

166

表 4.4：运算符优先级

结合律	和运算符	功能	用法	参考页码
左	::	全局作用域	::name	256
左	::	类作用域	class::name	79
左	::	命名空间作用域	namespace::name	74
左	.	成员选择	object.member	20
左	->	成员选择	pointer->member	98
左	[]	下标	expr[expr]	104
左	()	函数调用	name(expr_list)	20
左	()	类型构造	type(expr_list)	145
右	++	后置递增运算	lvalue++	131
右	--	后置递减运算	lvalue--	131
右	typeid	类型 ID	typeid(type)	731
右	typeid	运行时类型 ID	typeid(expr)	731
右	explicit cast	类型转换	*cast_name*<type>(expr)	144
右	++	前置递增运算	++lvalue	131
右	--	前置递减运算	--lvalue	131
右	~	位求反	~expr	136
右	!	逻辑非	!expr	126
右	-	一元负号	-expr	124
右	+	一元正号	+expr	124
右	*	解引用	*expr	48
右	&	取地址	&lvalue	47
右	()	类型转换	(type) expr	145
右	sizeof	对象的大小	sizeof expr	139

续表

结合律	和运算符	功能	用法	参考页码
右	sizeof	类型的大小	sizeof(type)	139
右	Sizeof...	参数包的大小	sizeof...(name)	619
右	new	创建对象	new type	407
右	new[]	创建数组	new type[size]	407
右	delete	释放对象	delete expr	409
右	delete[]	释放数组	delete[] expr	409
右	noexcept	能否抛出异常	noexcept (expr)	690
左	->*	指向成员选择的指针	ptr->*ptr_to_member	740
左	.*	指向成员选择的指针	obj.*ptr_to_member	740
左	*	乘法	expr * expr	124
左	/	除法	expr / expr	124
左	%	取模（取余）	expr % expr	124
左	+	加法	expr + expr	124
左	-	减法	expr - expr	124
左	<<	向左移位	expr << expr	136
左	>>	向右移位	expr >> expr	136
左	<	小于	expr < expr	126
左	<=	小于等于	expr <= expr	126
左	>	大于	expr > expr	126
左	>=	大于等于	expr >= expr	126
左	==	相等	expr == expr	126
左	!=	不相等	expr != expr	126
左	&	位与	expr & expr	136
左	^	位异或	expr ^ expr	136
左	|	位或	expr | expr	136
左	&&	逻辑与	expr && expr	126
左	||	逻辑或	expr || expr	126
右	? :	条件	expr ? expr : expr	134
右	=	赋值	lvalue = expr	129
右	*=, /=, %=	复合赋值	lvalue += expr 等	129
右	+=, -=			129
右	<<=, >>=			129
右	&=, |=, ^=			129
右	throw	抛出异常	throw expr	173
左	,	逗号	expr, expr	140

167

小结

168

C++语言提供了一套丰富的运算符，并定义了这些运算符作用于内置类型的运算对象时所执行的操作。此外，C++语言还支持运算符重载的机制，允许我们自己定义运算符作用于类类型时的含义。第 14 章将介绍如何定义作用于用户类型的运算符。

对于含有超过一个运算符的表达式，要想理解其含义关键要理解优先级、结合律和求值顺序。每个运算符都有其对应的优先级和结合律，优先级规定了复合表达式中运算符组合的方式，结合律则说明当运算符的优先级一样时应该如何组合。

大多数运算符并不明确规定运算对象的求值顺序：编译器有权自由选择先对左侧运算对象求值还是先对右侧运算对象求值。一般来说，运算对象的求值顺序对表达式的最终结果没有影响。但是，如果两个运算对象指向同一个对象而且其中一个改变了对象的值，就会导致程序出现不易发现的严重缺陷。

最后一点，运算对象经常从原始类型自动转换成某种关联的类型。例如，表达式中的小整型会自动提升成大整型。不论内置类型还是类类型都涉及类型转换的问题。如果需要，我们还可以显式地进行强制类型转换。

术语表

算术转换（arithmetic conversion）从一种算术类型转换成另一种算术类型。在二元运算符的上下文中，为了保留精度，算术转换通常把较小的类型转换成较大的类型（例如整型转换成浮点型）。

结合律（associativity）规定具有相同优先级的运算符如何组合在一起。结合律分为左结合律（运算符从左向右组合）和右结合律（运算符从右向左组合）。

二元运算符（binary operator）有两个运算对象参与运算的运算符。

强制类型转换（cast）一种显式的类型转换。

复合表达式（compound expression）含有多于一个运算符的表达式。

const_cast 一种涉及 `const` 的强制类型转换。将底层 `const` 对象转换成对应的非常量类型，或者执行相反的转换。

转换（conversion）一种类型的值改变成另一种类型的值的过程。C++语言定义了内置类型的转换规则。类类型同样可以转换。

dynamic_cast 和继承及运行时类型识别一起使用。参见 19.2 节（第 730 页）。

表达式（expression）C++程序中最低级别的计算。表达式将运算符作用于一个或多个运算对象，每个表达式都有对应的求值结果。表达式本身也可以作为运算对象，这时就得到了对多个运算符求值的复合表达式。

隐式转换（implicit conversion）由编译器自动执行的类型转换。假如表达式需要某种特定的类型而运算对象是另外一种类型，此时只要规则允许，编译器就会自动地将运算对象转换成所需的类型。

整型提升（integral promotion） 把一种较小的整数类型转换成与之最接近的较大整数类型的过程。不论是否真的需要，小整数类型（即 `short`、`char` 等）总是会得到提升。

169

左值（lvalue）是指那些求值结果为对象或函数的表达式。一个表示对象的非常量左值可以作为赋值运算符的左侧运算对象。

运算对象（operand）表达式在某些值上执行运算，这些值就是运算对象。一个运算

符有一个或多个相关的运算对象。

运算符（operator）决定表达式所做操作的符号。C++语言定义了一套运算符并说明了这些运算符作用于内置类型时的含义。C++还定义了运算符的优先级和结合律以及每种运算符处理的运算对象数量。可以重载运算符使其能处理类类型。

求值顺序（order of evaluation）是某个运算符的运算对象的求值顺序。大多数情况下，编译器可以任意选择运算对象求值的顺序。不过运算对象一定要在运算符之前得到求值结果。只有&&、||、条件和逗号四种运算符明确规定了求值顺序。

重载运算符（overloaded operator）针对某种运算符重新定义的适用于类类型的版本。第14章将介绍重载运算符的方法。

优先级（precedence）规定了复合表达式中不同运算符的执行顺序。与低优先级的运算符相比，高优先级的运算符组合得更紧密。

提升（promoted）参见整型提升。

reinterpret_cast 把运算对象的内容解释成另外一种类型。这种强制类型转换本质上依赖于机器而且非常危险。

结果（result）计算表达式得到的值或对象。

右值（rvalue）是指一种表达式，其结果是值而非值所在的位置。

短路求值（short-circuit evaluation） 是一个专有名词，描述逻辑与运算符和逻辑或运算符的执行过程。如果根据运算符的第一个运算对象就能确定整个表达式的结果，求值终止，此时第二个运算对象将不会被求值。

sizeof 是一个运算符，返回存储对象所需的字节数，该对象的类型可能是某个给定的类型名字，也可能由表达式的返回结果确定。

static_cast 显式地执行某种定义明确的类型转换，常用于替代由编译器隐式执行的类型转换。

一元运算符（unary operators）只有一个运算对象参与运算的运算符。

, 运算符（, operator）逗号运算符，是一种从左向右求值的二元运算符。逗号运算符的结果是右侧运算对象的值，当且仅当右侧运算对象是左值时逗号运算符的结果是左值。

? :运算符（?: operator）条件运算符，以下述形式提供 if-then-else 逻辑的表达式

```
cond ? expr1 : expr2;
```

如果条件 *cond* 为真，对 *expr1* 求值；否则对 *expr2* 求值。*expr1* 和 *expr2* 的类型应该相同或者能转换成同一种类型。*expr1* 和 *expr2* 中只有一个会被求值。

&&运算符（&& operator）逻辑与运算符，如果两个运算对象都是真，结果才为真。只有当左侧运算对象为真时才会检查右侧运算对象。

&运算符（& operator）位与运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位都是1，所得结果中该位为1；否则所得结果中该位为0。

170

^运算符（^ operator）位异或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位有且只有一个是1，所得结果中该位为1；否则所得结果中该位为0。

||运算符（|| operator）逻辑或运算符，任何一个运算对象是真，结果就为真。只有当左侧运算对象为假时才会检查右侧运算对象。

| 运算符（| operator）位或运算符，由两个运算对象生成一个新的整型值。如果两个运算对象对应的位至少有一个是1，所得结果中该位为1；否则所得结果中该位为0。

++运算符（++ operator）递增运算符。包括两种形式：前置版本和后置版本。前置递增运算符得到一个左值，它给运算符加1并得到运算对象改变后的值。后置递增运算符得到一个右值，它给运算符加1并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义+运算符，也会

有++运算符。

—运算符（— operator）递减运算符。包括两种形式：前置版本和后置版本。前置递减运算符得到一个左值，它从运算符减 1 并得到运算对象改变后的值。后置递减运算符得到一个右值，它从运算符减 1 并得到运算对象原始的、未改变的值的副本。注意：即使迭代器没有定义–运算符，也会有––运算符。

<<运算符（<< operator）左移运算符，将左侧运算对象的值的（可能是提升后的）副本向左移位，移动的位数由右侧运算对象确定。右侧运算对象必须大于等于 0 而且小于结果的位数。左侧运算对象应该是无符号类型，如果它是带符号类型，则一旦移动改变了符号位的值就会产生未定义的结果。

>>运算符（>> operator）右移运算符，除了移动方向相反，其他性质都和左移运算符类似。如果左侧运算对象是带符号类型，那么根据实现的不同新移入的内容也不同，新移入的位可能都是 0，也可能都是符号位的副本。

~运算符（~ operator）位求反运算符，生成一个新的整型值。该值的每一位恰好与（可能是提升后的）运算对象的对应位相反。

!运算符（! operator）逻辑非运算符，将它的运算对象的布尔值取反。如果运算对象是假，则结果为真，如果运算对象是真，则结果为假。

第5章
语句

内容

和大多数语言一样，C++提供了条件执行语句、重复执行相同代码的循环语句和用于中断当前控制流的跳转语句。本章将详细介绍 C++语言所支持的这些语句。

172 通常情况下，语句是顺序执行的。但除非是最简单的程序，否则仅有顺序执行远远不够。因此，C++语言提供了一组控制流（flow-of-control）语句以支持更复杂的执行路径。

5.1 简单语句

C++语言中的大多数语句都以分号结束，一个表达式，比如 `ival + 5`，末尾加上分号就变成了**表达式语句**（expression statement）。表达式语句的作用是执行表达式并丢弃掉求值结果：

```
ival + 5;          // 一条没什么实际用处的表达式语句
cout << ival;      // 一条有用的表达式语句
```

第一条语句没什么用处，因为虽然执行了加法，但是相加的结果没被使用。比较普遍的情况是，表达式语句中的表达式在求值时附带有其他效果，比如给变量赋了新值或者输出了结果。

空语句

最简单的语句是**空语句**（null statement），空语句中只含有一个单独的分号：

```
; // 空语句
```

如果在程序的某个地方，语法上需要一条语句但是逻辑上不需要，此时应该使用空语句。一种常见的情况是，当循环的全部工作在条件部分就可以完成时，我们通常会用到空语句。例如，我们想读取输入流的内容直到遇到一个特定的值为止，除此之外什么事情也不做：

```
// 重复读入数据直至到达文件末尾或某次输入的值等于 sought
while (cin >> s && s != sought)
    ; // 空语句
```

`while` 循环的条件部分首先从标准输入读取一个值并且隐式地检查 `cin`，判断读取是否成功。假定读取成功，条件的后半部分检查读进来的值是否等于 `sought` 的值。如果发现了想要的值，循环终止；否则，从 `cin` 中继续读取另一个值，再一次判断循环的条件。

Best Practices 使用空语句时应该加上注释，从而令读这段代码的人知道该语句是有意省略的。

别漏写分号，也别多写分号

因为空语句是一条语句，所以可用在任何允许使用语句的地方。由于这个原因，某些看起来非法的分号往往只不过是一条空语句而已，从语法上说得过去。下面的片段包含两条语句：表达式语句和空语句。

173
```
ival = v1 + v2;; // 正确：第二个分号表示一条多余的空语句
```

多余的空语句一般来说是无害的，但是如果在 `if` 或者 `while` 的条件后面跟了一个额外的分号就可能完全改变程序员的初衷。例如，下面的代码将无休止地循环下去：

```
// 出现了糟糕的情况：额外的分号，循环体是那条空语句
while (iter != svec.end()) ;      // while 循环体是那条空语句
    ++iter;                       // 递增运算不属于循环的一部分
```

虽然从形式上来看执行递增运算的语句前面有缩进，但它并不是循环的一部分。循环条件后面跟着的分号构成了一条空语句，它才是真正的循环体。

多余的空语句并非总是无害的。

复合语句（块）

复合语句（compound statement）是指用花括号括起来的（可能为空的）语句和声明的序列，复合语句也被称作**块**（block）。一个块就是一个作用域（参见 2.2.4 节，第 43 页），在块中引入的名字只能在块内部以及嵌套在块中的子块里访问。通常，名字在有限的区域内可见，该区域从名字定义处开始，到名字所在的（最内层）块的结尾为止。

如果在程序的某个地方，语法上需要一条语句，但是逻辑上需要多条语句，则应该使用复合语句。例如，while 或者 for 的循环体必须是一条语句，但是我们常常需要在循环体内做很多事情，此时就需要将多条语句用花括号括起来，从而把语句序列转变成块。

举个例子，回忆 1.4.1 节（第 10 页）的 while 循环：

```
while (val <= 10) {
    sum += val;       // 把 sum + val 的值赋给 sum。
    ++val;            // 给 val 加 1
}
```

程序从逻辑上来说要执行两条语句，但是 while 循环只能容纳一条。此时，把要执行的语句用花括号括起来，就将其转换成了一条（复合）语句。

块不以分号作为结束。

所谓空块，是指内部没有任何语句的一对花括号。空块的作用等价于空语句：

```
while (cin >> s && s != sought)
    { } // 空块
```

5.1 节练习

174

练习 5.1：什么是空语句？什么时候会用到空语句？

练习 5.2：什么是块？什么时候会用到块？

练习 5.3：使用逗号运算符（参见 4.10 节，第 140 页）重写 1.4.1 节（第 10 页）的 while 循环，使它不再需要块，观察改写之后的代码的可读性提高了还是降低了。

5.2 语句作用域

可以在 if、switch、while 和 for 语句的控制结构内定义变量。定义在控制结构当中的变量只在相应语句的内部可见，一旦语句结束，变量也就超出其作用范围了：

```
while (int i = get_num()) // 每次迭代时创建并初始化 i
    cout << i << endl;
i = 0; // 错误：在循环外部无法访问 i
```

如果其他代码也需要访问控制变量，则变量必须定义在语句的外部：

```
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

因为控制结构定义的对象的值马上要由结构本身使用，所以这些变量必须初始化。

5.2 节练习

练习 5.4：说明下列例子的含义，如果存在问题，试着修改它。

```
(a) while (string::iterator iter != s.end()) { /* ...*/ }
(b) while (bool status = find(word)) { /* ...*/ }
    if (!status) { /* ...*/ }
```

5.3 条件语句

C++语言提供了两种按条件执行的语句。一种是 `if` 语句，它根据条件决定控制流；另外一种是 `switch` 语句，它计算一个整型表达式的值，然后根据这个值从几条执行路径中选择一条。

5.3.1 if 语句

175

if 语句（if statement）的作用是：判断一个指定的条件是否为真，根据判断结果决定是否执行另外一条语句。if 语句包括两种形式：一种含有 else 分支，另外一种没有。简单 if 语句的语法形式是

```
if (condition)
    statement
```

`if else` 语句的形式是

```
if (condition)
    statement
else
    statement2
```

在这两个版本的 `if` 语句中，*condition* 都必须用圆括号包围起来。*condition* 可以是一个表达式，也可以是一个初始化了的变量声明（参见 5.2 节，第 155 页）。不管是表达式还是变量，其类型都必须能转换成（参见 4.11 节，第 141 页）布尔类型。通常情况下，*statement* 和 *statement2* 是块语句。

如果 *condition* 为真，执行 *statement*。当 *statement* 执行完成后，程序继续执行 `if` 语句后面的其他语句。

如果 *condition* 为假，跳过 *statement*。对于简单 `if` 语句来说，程序继续执行 `if` 语句后面的其他语句；对于 `if else` 语句来说，执行 *statement2*。

使用 if else 语句

我们举个例子来说明 if 语句的功能，程序的目的是把数字形式表示的成绩转换成字母形式。假设数字成绩的范围是从 0 到 100（包括 100 在内），其中 100 分对应的字母形式是“A++”，低于 60 分的成绩对应的字母形式是“F”。其他成绩每 10 个划分成一组：60 到 69（包括 69 在内）对应字母“D”、70 到 79 对应字母“C”，以此类推。使用 vector 对象存放字母成绩所有可能的取值：

```
const vector<string> scores = {"F", "D", "C", "B", "A", "A++"};
```

我们使用 if else 语句解决该问题，根据成绩是否合格执行不同的操作：

```
// 如果 grade 小于 60，对应的字母是 F；否则计算其下标
string lettergrade;
if (grade < 60)
    lettergrade = scores[0];
else
    lettergrade = scores[(grade - 50)/10];
```

判断 grade 的值是否小于 60，根据结果选择执行 if 分支还是 else 分支。在 else 分支中，由成绩计算得到一个下标，具体过程是：首先从 grade 中减去 50，然后执行整数除法（参见 4.2 节，在 125 页），去掉余数后所得的商就是数组 scores 对应的下标。

嵌套 if 语句

176

接下来让我们的程序更有趣点儿，试着给那些合格的成绩后面添加一个加号或减号。如果成绩的末位是 8 或者 9，添加一个加号；如果末位是 0、1 或 2，添加一个减号：

```
if (grade % 10 > 7)
    lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
else if (grade % 10 < 3)
    lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

我们使用取模运算符（参见 4.2 节，第 125 页）计算余数，根据余数决定添加哪种符号。

接着把这段添加符号的代码整合到转换成绩形式的代码中去：

```
// 如果成绩不合格，不需要考虑添加加号减号的问题
if (grade < 60)
    lettergrade = scores[0];
else {
    lettergrade = scores[(grade - 50)/10]; // 获得字母形式的成绩
    if (grade != 100) // 只要不是 A++，就考虑添加加号或减号
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
}
```

注意，我们使用花括号把第一个 else 后面的两条语句组合成了一个块。如果 grade 不小于 60 要做两件事：从数组 scores 中获取对应的字母成绩，然后根据条件设置加号或减号。

注意使用花括号

有一种常见的错误：本来程序中有几条语句应该作为一个块来执行，但是我们忘了用花括号把这些语句包围。在下面的例子中，添加加号减号的代码将被无条件地执行，这显然违背了我们的初衷：

```
if (grade < 60)
    lettergrade = scores[0];
else // 错误：缺少花括号
    lettergrade = scores[(grade - 50)/10];
    // 虽然下面的语句从形式上看有缩进，但是因为没有花括号，
    // 所以无论什么情况都会执行接下来的代码
    // 不及格的成绩也会添加上加号或减号，这显然是错误的
    if (grade != 100)
        if (grade % 10 > 7)
            lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
        else if (grade % 10 < 3)
            lettergrade += '-'; // 末尾是 0、1 或者 2 的成绩添加一个减号
```

要想发现这个错误可能非常困难，毕竟这段代码“看起来”是正确的。

为了避免此类问题，有些编码风格要求在 if 或 else 之后必须写上花括号（对 while 和 for 语句的循环体两端也有同样的要求）。这么做的好处是可以避免代码混乱不清，以后修改代码时如果想添加别的语句，也可以很容易地找到正确位置。

177

许多编辑器和开发环境都提供一种辅助工具，它可以自动地缩进代码以匹配其语法结构。善用此类工具益处多多。

悬垂 else

当一个 if 语句嵌套在另一个 if 语句内部时，很可能 if 分支会多于 else 分支。事实上，之前那个成绩转换的程序就有 4 个 if 分支，而只有 2 个 else 分支。这时候问题出现了：我们怎么知道某个给定的 else 是和哪个 if 匹配呢？

这个问题通常称作**悬垂 else**（dangling else），在那些既有 if 语句又有 if else 语句的编程语言中是个普遍存在的问题。不同语言解决该问题的思路也不同，就 C++而言，它规定 else 与离它最近的尚未匹配的 if 匹配，从而消除了程序的二义性。

当代码中 if 分支多于 else 分支时，程序员有时会感觉比较麻烦。举个例子来说明，对于添加加号减号的那个最内层的 if else 语句，我们用另外一组条件改写它：

```
// 错误：实际的执行过程并非像缩进格式显示的那样；else 分支匹配的是内层 if 语句
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; //末尾是 8 或者 9 的成绩添加一个加号
else
    lettergrade += '-'; //末尾是 3、4、5、6 或者 7 的成绩添加一个减号！
```

从代码的缩进格式来看，程序的初衷应该是希望 else 和外层的 if 匹配，也就是说，我们希望当 grade 的末位小于 3 时执行 else 分支。然而，不管我们是什么意图，也不管程序如何缩进，这里的 else 分支其实是内层 if 语句的一部分。最终，上面的代码将在末位大于 3 小于等于 7 的成绩后面添加减号！它的执行过程实际上等价于如下形式：

```
// 缩进格式与执行过程相符，但不是程序员的意图
if (grade % 10 >= 3)
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
    else
        lettergrade += '-'; // 末尾是 3、4、5、6 或者 7 的成绩添加一个减号!
```

使用花括号控制执行路径

要想使 else 分支和外层的 if 语句匹配起来，可以在内层 if 语句的两端加上花括号，使其成为一个块：

```
//末尾是 8 或者 9 的成绩添加一个加号，末尾是 0、1 或者 2 的成绩添加一个减号
if (grade % 10 >= 3) {
    if (grade % 10 > 7)
        lettergrade += '+'; // 末尾是 8 或者 9 的成绩添加一个加号
} else // 花括号强迫 else 与外层 if 匹配
    lettergrade += '-';       // 末尾是 0、1 或者 2 的成绩添加一个减号
```

语句属于块，意味着语句一定在块的边界之内，因此内层 if 语句在关键字 else 前面的那个花括号处已经结束了。else 不会再作为内层 if 的一部分。此时，最近的尚未匹配的 if 是外层 if，也就是我们希望 else 匹配的那个。 178

5.3.1 节练习

练习 5.5：写一段自己的程序，使用 if else 语句实现把数字成绩转换成字母成绩的要求。

练习 5.6：改写上一题的程序，使用条件运算符（参见 4.7 节，第 134 页）代替 if else 语句。

练习 5.7：改正下列代码段中的错误。

```
(a) if (ival1 != ival2)
        ival1 = ival2
    else ival1 = ival2 = 0;

(b) if (ival < minval)
        minval = ival;
        occurs = 1;
(c) if (int ival = get_value())
        cout << "ival = " << ival << endl;
    if (!ival)
        cout << "ival = 0\n";
(d) if (ival = 0)
        ival = get_value();
```

练习 5.8：什么是“悬垂 else”？C++语言是如何处理 else 子句的？

5.3.2 switch 语句

switch 语句（switch statement）提供了一条便利的途径使得我们能够在若干固定选项中做出选择。举个例子，假如我们想统计五个元音字母在文本中出现的次数，程序逻辑应该如下所示：

- 从输入的内容中读取所有字符。
- 令每一个字符都与元音字母的集合比较。
- 如果字符与某个元音字母匹配，将该字母的数量加 1。
- 显示结果。

例如，以（原书中）本章的文本作为输入内容，程序的输出结果将是：

```
Number of vowel a: 3195
Number of vowel e: 6230
Number of vowel i: 3102
Number of vowel o: 3289
Number of vowel u: 1033
```

179 要想实现这项功能，直接使用 switch 语句即可：

```
// 为每个元音字母初始化其计数值
unsigned aCnt = 0, eCnt = 0, iCnt = 0, oCnt = 0, uCnt = 0;
char ch;
while (cin >> ch) {
    // 如果 ch 是元音字母，将其对应的计数值加 1
    switch (ch) {
        case 'a':
            ++aCnt;
            break;
        case 'e':
            ++eCnt;
            break;
        case 'i':
            ++iCnt;
            break;
        case 'o':
            ++oCnt;
            break;
        case 'u':
            ++uCnt;
            break;
    }
}
// 输出结果
cout << "Number of vowel a: \t" << aCnt << '\n'
     << "Number of vowel e: \t" << eCnt << '\n'
     << "Number of vowel i: \t" << iCnt << '\n'
     << "Number of vowel o: \t" << oCnt << '\n'
     << "Number of vowel u: \t" << uCnt << endl;
```

switch 语句首先对括号里的表达式求值，该表达式紧跟在关键字 switch 的后面，可以是一个初始化的变量声明（参见 5.2 节，第 155 页）。表达式的值转换成整数类型，然后与每个 case 标签的值比较。

如果表达式和某个 case 标签的值匹配成功，程序从该标签之后的第一条语句开始执行，直到到达了 switch 的结尾或者是遇到一条 break 语句为止。

我们将在 5.5.1 节（第 170 页）详细介绍 break 语句，简言之，break 语句的作用是

中断当前的控制流。此例中，break 语句将控制权转移到 switch 语句外面。因为 switch 是 while 循环体内唯一的语句，所以从 switch 语句中断出来以后，程序的控制权将移到 while 语句的右花括号处。此时 while 语句内部没有其他语句要执行，所以 while 会返回去再一次判断条件是否满足。

如果 switch 语句的表达式和所有 case 都没有匹配上，将直接跳转到 switch 结构之后的第一条语句。刚刚说过，在上面的例子中，退出 switch 后控制权回到 while 语句的条件部分。

case 关键字和它对应的值一起被称为 **case 标签**（case label）。case 标签必须是整型常量表达式（参见 2.4.4 节，第 58 页）：

180

```
char ch = getVal();
int ival = 42;
switch(ch) {
case 3.14: // 错误：case 标签不是一个整数
case ival: // 错误：case 标签不是一个常量
//...
```

任何两个 case 标签的值不能相同，否则就会引发错误。另外，default 也是一种特殊的 case 标签，关于它的知识将在第 162 页介绍。

switch 内部的控制流

理解程序在 case 标签之间的执行流程非常重要。如果某个 case 标签匹配成功，将从该标签开始往后顺序执行所有 case 分支，除非程序显式地中断了这一过程，否则直到 switch 的结尾处才会停下来。要想避免执行后续 case 分支的代码，我们必须显式地告诉编译器终止执行过程。大多数情况下，在下一个 case 标签之前应该有一条 break 语句。

然而，也有一些时候默认的 switch 行为才是程序真正需要的。每个 case 标签只能对应一个值，但是有时候我们希望两个或更多个值共享同一组操作。此时，我们就故意省略掉 break 语句，使得程序能够连续执行若干个 case 标签。

例如，也许我们想统计的是所有元音字母出现的总次数：

```
unsigned vowelCnt = 0;
// ...
switch (ch)
{
    // 出现了 a、e、i、o 或 u 中的任意一个都会将 vowelCnt 的值加 1
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        ++vowelCnt;
        break;
}
```

在上面的代码中，几个 case 标签连写在一起，中间没有 break 语句。因此只要 ch 是元音字母，不管到底是五个中的哪一个都执行相同的代码。

C++程序的形式比较自由，所以 case 标签之后不一定非得换行。把几个 case 标签

写在一行里，强调这些 case 代表的是某个范围内的值：

```
switch (ch)
{
    // 另一种合法的书写形式
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
}
```

181

一般不要省略 case 分支最后的 break 语句。如果没写 break 语句，最好加一段注释说清楚程序的逻辑。

漏写 break 容易引发缺陷

有一种常见的错觉是程序只执行匹配成功的那个 case 分支的语句。例如，下面程序的统计结果是错误的：

```
// 警告：不正确的程序逻辑！
switch (ch) {
    case 'a':
        ++aCnt; // 此处应该有一条 break 语句
    case 'e':
        ++eCnt; // 此处应该有一条 break 语句
    case 'i':
        ++iCnt; // 此处应该有一条 break 语句
    case 'o':
        ++oCnt; // 此处应该有一条 break 语句
    case 'u':
        ++uCnt;
}
```

要想理解这段程序的执行过程，不妨假设 ch 的值是'e'。此时，程序直接执行 case 'e' 标签后面的代码，该代码把 eCnt 的值加 1。接下来，程序将跨越 case 标签的边界，接着递增 iCnt、oCnt 和 uCnt。

尽管 switch 语句不是非得在最后一个标签后面写上 break，但是为了安全起见，最好这么做。因为这样的话，即使以后再增加新的 case 分支，也不用再在前面补充 break 语句了。

default 标签

如果没有任何一个 case 标签能匹配上 switch 表达式的值，程序将执行紧跟在 **default 标签**（default label）后面的语句。例如，可以增加一个计数值来统计非元音字母的数量，只要在 default 分支内不断递增名为 otherCnt 的变量就可以了：

```
// 如果 ch 是一个元音字母，将相应的计数值加 1
switch (ch) {
    case 'a': case 'e': case 'i': case 'o': case 'u':
        ++vowelCnt;
        break;
    default:
```

```
            ++otherCnt;
            break;
        }
    }
```

在这个版本的程序中，如果 ch 不是元音字母，就从 default 标签开始执行并把 otherCnt 加 1。 182

即使不准备在 default 标签下做任何工作，定义一个 default 标签也是有用的。其目的在于告诉程序的读者，我们已经考虑到了默认的情况，只是目前什么也没做。

标签不应该孤零零地出现，它后面必须跟上一条语句或者另外一个 case 标签。如果 switch 结构以一个空的 default 标签作为结束，则该 default 标签后面必须跟上一条空语句或一个空块。

switch 内部的变量定义

如前所述，switch 的执行流程有可能会跨过某些 case 标签。如果程序跳转到了某个特定的 case，则 switch 结构中该 case 标签之前的部分会被忽略掉。这种忽略掉一部分代码的行为引出了一个有趣的问题：如果被略过的代码中含有变量的定义该怎么办？

答案是：如果在某处一个带有初值的变量位于作用域之外，在另一处该变量位于作用域之内，则从前一处跳转到后一处的行为是非法行为。

```
case true:
    // 因为程序的执行流程可能绕开下面的初始化语句，所以该 switch 语句不合法
    string file_name;     // 错误：控制流绕过一个隐式初始化的变量
    int ival = 0;         // 错误：控制流绕过一个显式初始化的变量
    int jval;             // 正确：因为 jval 没有初始化
    break;
case false:
    // 正确：jval 虽然在作用域内，但是它没有被初始化
    jval = next_num();        // 正确：给 jval 赋一个值
    if (file_name.empty())    // file_name 在作用域内，但是没有被初始化
        // ...
```

假设上述代码合法，则一旦控制流直接跳到 false 分支，也就同时略过了变量 file_name 和 ival 的初始化过程。此时这两个变量位于作用域之内，跟在 false 之后的代码试图在尚未初始化的情况下使用它们，这显然是行不通的。因此 C++语言规定，不允许跨过变量的初始化语句直接跳转到该变量作用域内的另一个位置。

如果需要为某个 case 分支定义并初始化一个变量，我们应该把变量定义在块内，从而确保后面的所有 case 标签都在变量的作用域之外。

```
case true:
    {
        // 正确：声明语句位于语句块内部
        string file_name = get_file_name();
        // ...
    }
    break;
case false:
```

```
if (file_name.empty()) // 错误：file_name 不在作用域之内
```

183

5.3.2 节练习

练习 5.9：编写一段程序，使用一系列 if 语句统计从 cin 读入的文本中有多少元音字母。

练习 5.10：我们之前实现的统计元音字母的程序存在一个问题：如果元音字母以大写形式出现，不会被统计在内。编写一段程序，既统计元音字母的小写形式，也统计大写形式，也就是说，新程序遇到'a'和'A'都应该递增 aCnt 的值，以此类推。

练习 5.11：修改统计元音字母的程序，使其也能统计空格、制表符和换行符的数量。

练习 5.12：修改统计元音字母的程序，使其能统计以下含有两个字符的字符序列的数量：ff、fl 和 fi。

练习 5.13：下面显示的每个程序都含有一个常见的编程错误，指出错误在哪里，然后修改它们。

(a)
```
unsigned aCnt = 0, eCnt = 0, iouCnt = 0;
char ch = next_text();
switch (ch) {
    case 'a': aCnt++;
    case 'e': eCnt++;
    default: iouCnt++;
}
```

(b)
```
unsigned index = some_value();
switch (index) {
    case 1:
        int ix = get_value();
        ivec[ ix ] = index;
        break;
    default:
        ix = ivec.size()-1;
        ivec[ ix ] = index;
}
```

(c)
```
unsigned evenCnt = 0, oddCnt = 0;
int digit = get_num() % 10;
switch (digit) {
    case 1, 3, 5, 7, 9:
        oddcnt++;
        break;
    case 2, 4, 6, 8, 10:
        evencnt++;
        break;
}
```

(d)
```
unsigned ival=512, jval=1024, kval=4096;
unsigned bufsize;
unsigned swt = get_bufCnt();
switch(swt) {
```

```
        case ival:
            bufsize = ival * sizeof(int);
            break;
        case jval:
            bufsize = jval * sizeof(int);
            break;
        case kval:
            bufsize = kval * sizeof(int);
            break;
}
```

5.4 迭代语句

迭代语句通常称为循环，它重复执行操作直到满足某个条件才停下来。while 和 for 语句在执行循环体之前检查条件，do while 语句先执行循环体，然后再检查条件。

5.4.1 while 语句

只要条件为真，**while 语句**（while statement）就重复地执行循环体，它的语法形式是

```
while (condition)
    statement
```

在 while 结构中，只要 *condition* 的求值结果为真就一直执行 *statement*（常常是一个块）。*condition* 不能为空，如果 *condition* 第一次求值就得 false，*statement* 一次都不执行。

while 的条件部分可以是一个表达式或者是一个带初始化的变量声明（参见 5.2 节，第 155 页）。通常来说，应该由条件本身或者是循环体设法改变表达式的值，否则循环可能无法终止。

定义在 while 条件部分或者 while 循环体内的变量每次迭代都经历从创建到销毁的过程。

使用 while 循环

当不确定到底要迭代多少次时，使用 while 循环比较合适，比如读取输入的内容就是如此。还有一种情况也应该使用 while 循环，这就是我们想在循环结束后访问循环控制变量。例如：

185

```
vector<int> v;
int i;
// 重复读入数据，直至到达文件末尾或者遇到其他输入问题
while (cin >> i)
    v.push_back(i);
// 寻找第一个负值元素
auto beg = v.begin();
while (beg != v.end() && *beg >= 0)
    ++beg;
if (beg == v.end())
    // 此时我们知道 v 中的所有元素都大于等于 0
```

第一个循环从标准输入中读取数据，我们一开始不清楚循环要执行多少次，当 cin 读取到无效数据、遇到其他一些输入错误或是到达文件末尾时循环条件失效。第二个循环重复执行直到遇到一个负值为止，循环终止后，beg 或者等于 v.end()，或者指向 v 中一个小于 0 的元素。可以在 while 循环外继续使用 beg 的状态以进行其他处理。

5.4.1 节练习

练习 5.14：编写一段程序，从标准输入中读取若干 string 对象并查找连续重复出现的单词。所谓连续重复出现的意思是：一个单词后面紧跟着这个单词本身。要求记录连续重复出现的最大次数以及对应的单词。如果这样的单词存在，输出重复出现的最大次数；如果不存在，输出一条信息说明任何单词都没有连续出现过。例如，如果输入是

```
how now now now brown cow cow
```

那么输出应该表明单词 now 连续出现了 3 次。

5.4.2 传统的 for 语句

for 语句的语法形式是

```
for (init-statemen; condition; expression)
    statement
```

关键字 for 及括号里的部分称作 for 语句头。

init-statement 必须是以下三种形式中的一种：声明语句、表达式语句或者空语句，因为这些语句都以分号作为结束，所以 for 语句的语法形式也可以看做

```
for (initializer; condition; expression)
    statement
```

一般情况下，*init-statement* 负责初始化一个值，这个值将随着循环的进行而改变。*condition* 作为循环控制的条件，只要 *condition* 为真，就执行一次 *statement*。如果 *condition* 第一次的求值结果就是 false，则 *statement* 一次也不会执行。*expression* 负责修改 *init-statement* 初始化的变量，这个变量正好就是 *condition* 检查的对象，修改发生在每次循环迭代之后。*statement* 可以是一条单独的语句也可以是一条复合语句。

186

传统 for 循环的执行流程

我们以 3.2.3 节（第 85 页）的 for 循环为例：

```
// 重复处理 s 中的字符直至我们处理完全部字符或者遇到了一个表示空白的字符
for (decltype(s.size()) index = 0;
     index != s.size() && !isspace(s[index]); ++index)
         s[index] = toupper(s[index]); // 将当前字符改成大写形式
```

求值的顺序如下所示：

1. 循环开始时，首先执行一次 *init-statement*。此例中，定义 index 并初始化为 0。
2. 接下来判断 *condition*。如果 index 不等于 s.size() 而且在 s[index] 位置的字符不是空白，则执行 for 循环体的内容。否则，循环终止。如果第一次迭代时条件就为假，for 循环体一次也不会执行。
3. 如果条件为真，执行循环体。此例中，for 循环体将 s[index] 位置的字符改写

成大写形式。

4. 最后执行 *expression*。此例中，将 index 的值加 1。

这 4 步说明了 for 循环第一次迭代的过程。其中第 1 步只在循环开始时执行一次，第 2、3、4 步重复执行直到条件为假时终止，也就是在 s 中遇到一个空白字符或者 index 大于 s.size() 时终止。

牢记 for 语句头中定义的对象只在 for 循环体内可见。因此在上面的例子中，for 循环结束后 index 就不可用了。

for 语句头中的多重定义

和其他的声明一样，*init-statement* 也可以定义多个对象。但是 *init-statement* 只能有一条声明语句，因此，所有变量的基础类型必须相同（参见 2.3 节，第 45 页）。举个例子，我们用下面的循环把 vector 的元素拷贝一份添加到原来的元素后面：

```
// 记录下 v 的大小，当到达原来的最后一个元素后结束循环
for (decltype(v.size()) i = 0, sz = v.size(); i != sz; ++i)
    v.push_back(v[i]);
```

在这个循环中，我们在 *init-statement* 里同时定义了索引 i 和循环控制变量 sz。

省略 for 语句头的某些部分

187

for 语句头能省略掉 *init-statement*、*condition* 和 *expression* 中的任何一个（或者全部）。

如果无须初始化，则我们可以使用一条空语句作为 *init-statement*。例如，对于在 vector 对象中寻找第一个负数的程序，完全能用 for 循环改写：

```
auto beg = v.begin();
for ( /* 空语句 */; beg != v.end() && *beg >= 0; ++beg)
    ; // 什么也不做
```

注意，分号必须保留以表明我们省略掉了 *init-statement*。说得更准确一点，分号表示的是一个空的 *init-statement*。在这个循环中，因为所有要做的工作都在 for 语句头的条件和表达式部分完成了，所以 for 循环体也是空的。其中，条件部分决定何时停止查找，表达式部分递增迭代器。

省略 *condition* 的效果等价于在条件部分写了一个 true。因为条件的值永远是 true，所以在循环体内必须有语句负责退出循环，否则循环就会无休止地执行下去：

```
for (int i = 0; /* 条件为空 */ ; ++i) {
    // 对 i 进行处理，循环内部的代码必须负责终止迭代过程！
}
```

我们也能省略掉 for 语句头中的 *expression*，但是在这样的循环中就要求条件部分或者循环体必须改变迭代变量的值。举个例子，之前有一个将整数读入 vector 的 while 循环，我们使用 for 语句改写它：

```
vector<int> v;
for (int i; cin >> i; /* 表达式为空 */ )
    v.push_back(i);
```

因为条件部分能改变 i 的值，所以这个循环无须表达式部分。其中，条件部分不断检查输

入流的内容，只要读取完所有的输入或者遇到一个输入错误就终止循环。

188

5.4.2 节练习

练习 5.15：说明下列循环的含义并改正其中的错误。

```
(a) for (int ix = 0; ix != sz; ++ix) { /* ...*/ }
    if (ix != sz)
        // ...
(b) int ix;
    for (ix != sz; ++ix) { /* ...*/ }
(c) for (int ix = 0; ix != sz; ++ix, ++ sz) { /* ...*/ }
```

练习 5.16：while 循环特别适用于那种条件保持不变、反复执行操作的情况，例如，当未达到文件末尾时不断读取下一个值。for 循环则更像是在按步骤迭代，它的索引值在某个范围内依次变化。根据每种循环的习惯用法各自编写一段程序，然后分别用另一种循环改写。如果只能使用一种循环，你倾向于使用哪种呢？为什么？

练习 5.17：假设有两个包含整数的 vector 对象，编写一段程序，检验其中一个 vector 对象是否是另一个的前缀。为了实现这一目标，对于两个不等长的 vector 对象，只需挑出长度较短的那个，把它的所有元素和另一个 vector 对象比较即可。例如，如果两个 vector 对象的元素分别是 0、1、1、2 和 0、1、1、2、3、5、8，则程序的返回结果应该为真。

5.4.3 范围 for 语句

C++11

C++11 新标准引入了一种更简单的 for 语句，这种语句可以遍历容器或其他序列的所有元素。**范围 for 语句**（range for statement）的语法形式是：

```
for (declaration : expression)
    statement
```

expression 表示的必须是一个序列，比如用花括号括起来的初始值列表（参见 3.3.1 节，第 88 页）、数组（参见 3.5 节，第 101 页）或者 vector 或 string 等类型的对象，这些类型的共同特点是拥有能返回迭代器的 begin 和 end 成员（参见 3.4 节，第 95 页）。

declaration 定义一个变量，序列中的每个元素都得能转换成该变量的类型（参见 4.11 节，第 141 页）。确保类型相容最简单的办法是使用 auto 类型说明符（参见 2.5.2 节，第 61 页），这个关键字可以令编译器帮助我们指定合适的类型。如果需要对序列中的元素执行写操作，循环变量必须声明成引用类型。

每次迭代都会重新定义循环控制变量，并将其初始化成序列中的下一个值，之后才会执行 *statement*。像往常一样，*statement* 可以是一条单独的语句也可以是一个块。所有元素都处理完毕后循环终止。

之前我们已经接触过几个这样的循环。接下来的例子将把 vector 对象中的每个元素都翻倍，它涵盖了范围 for 语句的几乎所有语法特征：

```
vector<int> v = {0,1,2,3,4,5,6,7,8,9};
// 范围变量必须是引用类型，这样才能对元素执行写操作
for (auto &r : v)     // 对于 v 中的每一个元素
```

```
    r *= 2;          // 将 v 中每个元素的值翻倍
```

for 语句头声明了循环控制变量 r，并把它和 v 关联在一起，我们使用关键字 auto 令编译器为 r 指定正确的类型。由于准备修改 v 的元素的值，因此将 r 声明成引用类型。此时，在循环体内给 r 赋值，即改变了 r 所绑定的元素的值。

范围 for 语句的定义来源于与之等价的传统 for 语句：

```
for (auto beg = v.begin(), end = v.end(); beg != end; ++beg) {
    auto &r = *beg;  // r 必须是引用类型，这样才能对元素执行写操作
    r *= 2;          // 将 v 中每个元素的值翻倍
}
```

学习了范围 for 语句的原理之后，我们也就不难理解为什么在 3.3.2 节（第 90 页）强调不能通过范围 for 语句增加 vector 对象（或者其他容器）的元素了。在范围 for 语句中，预存了 end() 的值。一旦在序列中添加（删除）元素，end 函数的值就可能变得无效了（参见 3.4.1 节，第 98 页）。关于这一点，将在 9.3.6 节（第 315 页）做更详细的介绍。 189

5.4.4 do while 语句

do while 语句（do while statement）和 while 语句非常相似，唯一的区别是，do while 语句先执行循环体后检查条件。不管条件的值如何，我们都至少执行一次循环。do while 语句的语法形式如下所示：

```
do
    statement
while (condition);
```

do while 语句应该在括号包围起来的条件后面用一个分号表示语句结束。

在 do 语句中，求 *condition* 的值之前首先执行一次 *statement*，*condition* 不能为空。如果 *condition* 的值为假，循环终止；否则，重复循环过程。*condition* 使用的变量必须定义在循环体之外。

我们可以使用 do while 循环（不断地）执行加法运算：

```
// 不断提示用户输入一对数，然后求其和
string rsp; // 作为循环的条件，不能定义在 do 的内部
do {
    cout << "please enter two values: ";
    int val1 = 0, val2 = 0;
    cin >> val1 >> val2;
    cout << "The sum of " << val1 << " and " << val2
         << " = " << val1 + val2 << "\n\n"
         << "More? Enter yes or no: ";
    cin  >> rsp;
} while (!rsp.empty() && rsp[0] != 'n');
```

循环首先提示用户输入两个数字，然后输出它们的和并询问用户是否继续。条件部分检查用户做出的回答，如果用户没有回答，或者用户的回答以字母 n 开始，循环都将终止。否则循环继续执行。

因为对于 do while 来说先执行语句或者块，后判断条件，所以不允许在条件部分

定义变量：

```
do {
    // ...
    mumble(foo);
} while (int foo = get_foo()); // 错误：将变量声明放在了 do 的条件部分
```

如果允许在条件部分定义变量，则变量的使用出现在定义之前，这显然是不合常理的！

190

5.4.4 节练习

练习 5.18：说明下列循环的含义并改正其中的错误。

(a) do

```
        int v1, v2;
        cout << "Please enter two numbers to sum:" ;
        if (cin >> v1 >> v2)
            cout << "Sum is: " << v1 + v2 << endl;
    while (cin);
```

(b) do {

```
        // ...
    } while (int ival = get_response());
```

(c) do {

```
        int ival = get_response();
    } while (ival);
```

练习 5.19：编写一段程序，使用 do while 循环重复地执行下述任务：首先提示用户输入两个 string 对象，然后挑出较短的那个并输出它。

5.5 跳转语句

跳转语句中断当前的执行过程。C++语言提供了 4 种跳转语句：break、continue、goto 和 return。本章介绍前三种跳转语句，return 语句将在 6.3 节（第 199 页）进行介绍。

5.5.1 break 语句

break 语句（break statement）负责终止离它最近的 while、do while、for 或 switch 语句，并从这些语句之后的第一条语句开始继续执行。

break 语句只能出现在迭代语句或者 switch 语句内部（包括嵌套在此类循环里的语句或块的内部）。break 语句的作用范围仅限于最近的循环或者 switch：

```
string buf;
while (cin >> buf && !buf.empty()) {
    switch(buf[0]) {
    case '-':
        // 处理到第一个空白为止
        for (auto it = buf.begin()+1; it != buf.end(); ++it) {
            if (*it == ' ')
                break; // #1，离开 for 循环
            // ...
```

```
            }
            // break #1 将控制权转移到这里
            // 剩余的'-'处理:
            break; // #2, 离开 switch 语句
            case '+':
                //...
        } // 结束 switch
        // 结束 switch: break #2 将控制权转移到这里
    } // 结束 while
```

191

标记为#1 的 break 语句负责终止连字符 case 标签后面的 for 循环。它不但不会终止 switch 语句，甚至连当前的 case 分支也终止不了。接下来，程序继续执行 for 循环之后的第一条语句，这条语句可能接着处理连字符的情况，也可能是另一条用于终止当前分支的 break 语句。

标记为#2 的 break 语句负责终止 switch 语句，但是不能终止 while 循环。执行完这个 break 后，程序继续执行 while 的条件部分。

5.5.1 节练习

练习 5.20：编写一段程序，从标准输入中读取 string 对象的序列直到连续出现两个相同的单词或者所有单词都读完为止。使用 while 循环一次读取一个单词，当一个单词连续出现两次时使用 break 语句终止循环。输出连续重复出现的单词，或者输出一个消息说明没有任何单词是连续重复出现的。

5.5.2 continue 语句

continue 语句（continue statement）终止最近的循环中的当前迭代并立即开始下一次迭代。continue 语句只能出现在 for、while 和 do while 循环的内部，或者嵌套在此类循环里的语句或块的内部。和 break 语句类似的是，出现在嵌套循环中的 continue 语句也仅作用于离它最近的循环。和 break 语句不同的是，只有当 switch 语句嵌套在迭代语句内部时，才能在 switch 里使用 continue。

continue 语句中断当前的迭代，但是仍然继续执行循环。对于 while 或者 do while 语句来说，继续判断条件的值；对于传统的 for 循环来说，继续执行 for 语句头的 *expression*；而对于范围 *for* 语句来说，则是用序列中的下一个元素初始化循环控制变量。

例如，下面的程序每次从标准输入中读取一个单词。循环只对那些以下画线开头的单词感兴趣，其他情况下，我们直接终止当前的迭代并获取下一个单词：

```
string buf;
while (cin >> buf && !buf.empty()) {
    if (buf[0] != '_')
        continue; // 接着读取下一个输入
    // 程序执行过程到了这里？说明当前的输入是以下画线开始的；接着处理 buf……
}
```

5.5.2 节练习

192

练习 5.21：修改 5.5.1 节（第 171 页）练习题的程序，使其找到的重复单词必须以大写字母开头。

5.5.3 goto 语句

goto 语句（goto statement）的作用是从 goto 语句无条件跳转到同一函数内的另一条语句。

Best Practices 不要在程序中使用 goto 语句，因为它使得程序既难理解又难修改。

goto 语句的语法形式是

```
goto label;
```

其中，*label* 是用于标识一条语句的标示符。**带标签语句**（labeled statement）是一种特殊的语句，在它之前有一个标示符以及一个冒号：

```
end: return; // 带标签语句，可以作为 goto 的目标
```

标签标示符独立于变量或其他标示符的名字，因此，标签标示符可以和程序中其他实体的标示符使用同一个名字而不会相互干扰。goto 语句和控制权转向的那条带标签的语句必须位于同一个函数之内。

和 switch 语句类似，goto 语句也不能将程序的控制权从变量的作用域之外转移到作用域之内：

```
    // ...
    goto end;
    int ix = 10; // 错误：goto 语句绕过了一个带初始化的变量定义
end:
    // 错误：此处的代码需要使用 ix，但是 goto 语句绕过了它的声明
    ix = 42;
```

向后跳过一个已经执行的定义是合法的。跳回到变量定义之前意味着系统将销毁该变量，然后重新创建它：

```
// 向后跳过一个带初始化的变量定义是合法的
begin:
    int sz = get_size();
    if (sz <= 0) {
        goto begin;
    }
```

在上面的代码中，goto 语句执行后将销毁 sz。因为跳回到 begin 的动作跨过了 sz 的定义语句，所以 sz 将重新定义并初始化。

193

5.5.3 节练习

练习 5.22：本节的最后一个例子跳回到 begin，其实使用循环能更好地完成该任务。重写这段代码，注意不再使用 goto 语句。

5.6 try 语句块和异常处理

异常是指存在于运行时的反常行为，这些行为超出了函数正常功能的范围。典型的异常包括失去数据库连接以及遇到意外输入等。处理反常行为可能是设计所有系统最难的一部分。

当程序的某部分检测到一个它无法处理的问题时，需要用到异常处理。此时，检测出问题的部分应该发出某种信号以表明程序遇到了故障，无法继续下去了，而且信号的发出方无须知道故障将在何处得到解决。一旦发出异常信号，检测出问题的部分也就完成了任务。

如果程序中含有可能引发异常的代码，那么通常也会有专门的代码处理问题。例如，如果程序的问题是输入无效，则异常处理部分可能会要求用户重新输入正确的数据；如果丢失了数据库连接，会发出报警信息。

异常处理机制为程序中异常检测和异常处理这两部分的协作提供支持。在 C++语言中，异常处理包括：

- **throw 表达式**（throw expression），异常检测部分使用 throw 表达式来表示它遇到了无法处理的问题。我们说 throw **引发**（raise）了异常。
- **try 语句块**（try block），异常处理部分使用 try 语句块处理异常。try 语句块以关键字 try 开始，并以一个或多个 **catch 子句**（catch clause）结束。try 语句块中代码抛出的异常通常会被某个 catch 子句处理。因为 catch 子句“处理”异常，所以它们也被称作**异常处理代码**（exception handler）。
- 一套**异常类**（exception class），用于在 throw 表达式和相关的 catch 子句之间传递异常的具体信息。

在本节的剩余部分，我们将分别介绍异常处理的这三个组成部分。在 18.1 节（第 684 页）还将介绍更多关于异常的知识。

5.6.1 throw 表达式

程序的异常检测部分使用 throw 表达式引发一个异常。throw 表达式包含关键字 throw 和紧随其后的一个表达式，其中表达式的类型就是抛出的异常类型。throw 表达式后面通常紧跟一个分号，从而构成一条表达式语句。

举个简单的例子，回忆 1.5.2 节（第 20 页）把两个 Sales_item 对象相加的程序。194 这个程序检查它读入的记录是否是关于同一种书籍的，如果不是，输出一条信息然后退出。

```
Sales_item item1, item2;
cin >> item1 >> item2;
// 首先检查 item1 和 item2 是否表示同一种书籍
if (item1.isbn() == item2.isbn()) {
    cout << item1 + item2 << endl;
    return 0; // 表示成功
} else {
    cerr << "Data must refer to same ISBN" << endl;
    return -1; // 表示失败
}
```

在真实的程序中，应该把对象相加的代码和用户交互的代码分离开来。此例中，我们改写程序使得检查完成后不再直接输出一条信息，而是抛出一个异常：

```
// 首先检查两条数据是否是关于同一种书籍的
if (item1.isbn() != item2.isbn())
    throw runtime_error("Data must refer to same ISBN");
// 如果程序执行到了这里，表示两个 ISBN 是相同的
cout << item1 + item2 << endl;
```

在这段代码中，如果 ISBN 不一样就抛出一个异常，该异常是类型 runtime_error 的对象。抛出异常将终止当前的函数，并把控制权转移给能处理该异常的代码。

类型 runtime_error 是标准库异常类型的一种，定义在 stdexcept 头文件中。关于标准库异常类型更多的知识将在 5.6.3 节（第 176 页）介绍。我们必须初始化 runtime_error 的对象，方式是给它提供一个 string 对象或者一个 C 风格的字符串（参见 3.5.4 节，第 109 页），这个字符串中有一些关于异常的辅助信息。

5.6.2 try 语句块

try 语句块的通用语法形式是

```
try {
    program-statements
} catch (exception-declaration) {
    handler-statements
} catch (exception-declaration) {
    handler-statements
} // ...
```

try 语句块的一开始是关键字 try，随后紧跟着一个块，这个块就像大多数时候那样是花括号括起来的语句序列。

195 跟在 try 块之后的是一个或多个 catch 子句。catch 子句包括三部分：关键字 catch、括号内一个（可能未命名的）对象的声明（称作**异常声明**，exception declaration）以及一个块。当选中了某个 catch 子句处理异常之后，执行与之对应的块。catch 一旦完成，程序跳转到 try 语句块最后一个 catch 子句之后的那条语句继续执行。

try 语句块中的 *program-statements* 组成程序的正常逻辑，像其他任何块一样，*program-statements* 可以有包括声明在内的任意 C++语句。一如往常，try 语句块内声明的变量在块外部无法访问，特别是在 catch 子句内也无法访问。

编写处理代码

在之前的例子里，我们使用了一个 throw 表达式以避免把两个代表不同书籍的 Sales_item 相加。我们假设执行 Sales_item 对象加法的代码是与用户交互的代码分离开来的。其中与用户交互的代码负责处理发生的异常，它的形式可能如下所示：

```
while (cin >> item1 >> item2) {
    try {
        // 执行添加两个 Sales_item 对象的代码
        // 如果添加失败，代码抛出一个 runtime_error 异常
    } catch (runtime_error err) {
        // 提醒用户两个 ISBN 必须一致，询问是否重新输入
        cout << err.what()
             << "\nTry Again? Enter y or n" << endl;
        char c;
        cin >> c;
```

```
            if (!cin || c == 'n')
                break; // 跳出 while 循环
        }
    }
```

程序本来要执行的任务出现在 try 语句块中，这是因为这段代码可能会抛出一个 runtime_error 类型的异常。

try 语句块对应一个 catch 子句，该子句负责处理类型为 runtime_error 的异常。如果 try 语句块的代码抛出了 runtime_error 异常，接下来执行 catch 块内的语句。在我们书写的 catch 子句中，输出一段提示信息要求用户指定程序是否继续。如果用户输入'n'，执行 break 语句并退出 while 循环；否则，直接执行 while 循环的右侧花括号，意味着程序控制权跳回到 while 条件部分准备下一次迭代。

给用户的提示信息中输出了 err.what() 的返回值。我们知道 err 的类型是 runtime_error，因此能推断 what 是 runtime_error 类的一个成员函数（参见 1.5.2 节，第 20 页）。每个标准库异常类都定义了名为 what 的成员函数，这些函数没有参数，返回值是 C 风格字符串（即 const char*）。其中，runtime_error 的 what 成员返回的是初始化一个具体对象时所用的 string 对象的副本。如果上一节编写的代码抛出异常，则本节的 catch 子句输出

196

```
Data must refer to same ISBN
Try Again? Enter y or n
```

函数在寻找处理代码的过程中退出

在复杂系统中，程序在遇到抛出异常的代码前，其执行路径可能已经经过了多个 try 语句块。例如，一个 try 语句块可能调用了包含另一个 try 语句块的函数，新的 try 语句块可能调用了包含又一个 try 语句块的新函数，以此类推。

寻找处理代码的过程与函数调用链刚好相反。当异常被抛出时，首先搜索抛出该异常的函数。如果没找到匹配的 catch 子句，终止该函数，并在调用该函数的函数中继续寻找。如果还是没有找到匹配的 catch 子句，这个新的函数也被终止，继续搜索调用它的函数。以此类推，沿着程序的执行路径逐层回退，直到找到适当类型的 catch 子句为止。

如果最终还是没能找到任何匹配的 catch 子句，程序转到名为 **terminate** 的标准库函数。该函数的行为与系统有关，一般情况下，执行该函数将导致程序非正常退出。

对于那些没有任何 try 语句块定义的异常，也按照类似的方式处理：毕竟，没有 try 语句块也就意味着没有匹配的 catch 子句。如果一段程序没有 try 语句块且发生了异常，系统会调用 terminate 函数并终止当前程序的执行。

提示：编写异常安全的代码非常困难

要好好理解这句话：异常中断了程序的正常流程。异常发生时，调用者请求的一部分计算可能已经完成了，另一部分则尚未完成。通常情况下，略过部分程序意味着某些对象处理到一半就戛然而止，从而导致对象处于无效或未完成的状态，或者资源没有正常释放，等等。那些在异常发生期间正确执行了“清理”工作的程序被称作异常安全（exception safe）的代码。然而经验表明，编写异常安全的代码非常困难，这部分知识也（远远）超出了本书的范围。

对于一些程序来说，当异常发生时只是简单地终止程序。此时，我们不怎么需要担

心异常安全的问题。

但是对于那些确实要处理异常并继续执行的程序，就要加倍注意了。我们必须时刻清楚异常何时发生，异常发生后程序应如何确保对象有效、资源无泄漏、程序处于合理状态，等等。

我们会在本书中介绍一些比较常规的提升异常安全性的技术。但是读者需要注意，如果你的程序要求非常鲁棒的异常处理，那么仅有我们介绍的这些技术恐怕还是不够的。

197

5.6.3 标准异常

C++标准库定义了一组类，用于报告标准库函数遇到的问题。这些异常类也可以在用户编写的程序中使用，它们分别定义在4个头文件中：

- exception 头文件定义了最通用的异常类 exception。它只报告异常的发生，不提供任何额外信息。
- stdexcept 头文件定义了几种常用的异常类，详细信息在表5.1中列出。
- new 头文件定义了 bad_alloc 异常类型，这种类型将在12.1.2节（第407页）详细介绍。
- type_info 头文件定义了 bad_cast 异常类型，这种类型将在19.2节（第731页）详细介绍。

表5.1：<stdexcept>定义的异常类

exception	最常见的问题
runtime_error	只有在运行时才能检测出的问题
range_error	运行时错误：生成的结果超出了有意义的值域范围
overflow_error	运行时错误：计算上溢
underflow_error	运行时错误：计算下溢
logic_error	程序逻辑错误
domain_error	逻辑错误：参数对应的结果值不存在
invalid_argument	逻辑错误：无效参数
length_error	逻辑错误：试图创建一个超出该类型最大长度的对象
out_of_range	逻辑错误：使用一个超出有效范围的值

标准库异常类只定义了几种运算，包括创建或拷贝异常类型的对象，以及为异常类型的对象赋值。

我们只能以默认初始化（参见2.2.1节，第40页）的方式初始化 exception、bad_alloc 和 bad_cast 对象，不允许为这些对象提供初始值。

其他异常类型的行为则恰好相反：应该使用 string 对象或者C风格字符串初始化这些类型的对象，但是不允许使用默认初始化的方式。当创建此类对象时，必须提供初始值，该初始值含有错误相关的信息。

异常类型只定义了一个名为 what 的成员函数，该函数没有任何参数，返回值是一个指向C风格字符串（参见3.5.4节，第109页）的 const char*。该字符串的目的是提供关于异常的一些文本信息。

what 函数返回的 C 风格字符串的内容与异常对象的类型有关。如果异常类型有一个字符串初始值，则 what 返回该字符串。对于其他无初始值的异常类型来说，what 返回的内容由编译器决定。 198

5.6.3 节练习

练习 5.23：编写一段程序，从标准输入读取两个整数，输出第一个数除以第二个数的结果。

练习 5.24：修改你的程序，使得当第二个数是 0 时抛出异常。先不要设定 catch 子句，运行程序并真的为除数输入 0，看看会发生什么？

练习 5.25：修改上一题的程序，使用 try 语句块去捕获异常。catch 子句应该为用户输出一条提示信息，询问其是否输入新数并重新执行 try 语句块的内容。

199

小结

C++语言仅提供了有限的语句类型，它们中的大多数会影响程序的控制流程：

- while、for 和 do while 语句，执行迭代操作。
- if 和 switch 语句，提供条件分支结构。
- continue 语句，终止循环的当前一次迭代。
- break 语句，退出循环或者 switch 语句。
- goto 语句，将控制权转移到一条带标签的语句。
- try 和 catch，将一段可能抛出异常的语句序列括在花括号里构成 try 语句块。catch 子句负责处理代码抛出的异常。
- throw 表达式语句，存在于代码块中，将控制权转移到相关的 catch 子句。
- return 语句，终止函数的执行。我们将在第 6 章介绍 return 语句。

除此之外还有表达式语句和声明语句。表达式语句用于求解表达式，关于变量的声明和定义在第 2 章已经介绍过了。

术语表

块（block）包围在花括号内的由 0 条或多条语句组成的序列。块也是一条语句，所以只要是能使用语句的地方，就可以使用块。

break 语句（break statement）终止离它最近的循环或 switch 语句。控制权转移到循环或 switch 之后的第一条语句。

case 标签（case label）在 switch 语句中紧跟在 case 关键字之后的常量表达式（参见2.4.4节，第58页）。在同一个 switch 语句中任意两个 case 标签的值不能相同。

catch 子句（catch clause）由三部分组成：catch 关键字、括号里的异常声明以及一个语句块。catch 子句的代码负责处理在异常声明中定义的异常。

复合语句（compound statement）和块是同义词。

continue 语句（continue statement）终止离它最近的循环的当前迭代。控制权转移到 while 或 do while 语句的条件部分、或者范围 for 循环的下一次迭代、或者传统 for 循环头部的表达式。

悬垂 else（dangling else）是一个俗语，指的是如何处理嵌套 if 语句中 if 分支多于 else 分支的情况。C++语言规定，else 应该与前一个未匹配的 if 匹配在一起。使用花括号可以把位于内层的 if 语句隐藏起来，这样程序员就能更好地控制 else 该与哪个 if 匹配。

default 标签（default label）是一种特殊的 case 标签，当 switch 表达式的值与所有 case 标签都无法匹配时，程序执行 default 标签下的内容。

200 **do while 语句**（do while statement）与 while 语句类似，区别是 do while 语句先执行循环体，再判断条件。循环体代码至少会执行一次。

异常类（exception class）是标准库定义的一组类，用于表示程序发生的错误。表 5.1（第 176 页）列出了不同用途的异常类。

异常声明（exception declaration）位于 catch 子句中的声明，指定了该 catch 子句能处理的异常类型。

异常处理代码（exception handler）程序某处引发异常后，用于处理该异常的另一